Report from the Fall 1999 Embedded Systems Conference

November 17, 1999

Compilations of notes by attendees from the Jet Propulsion Laboratory, California Institute of Technology. Specifically,

By the members of the:

Realtime Interferometer Software Group, Section 383
Graham Hardy
Brad Hines
Phil Irwin
Beth McKenney
Martin Regehr

Guidance and Control Analysis Group, Section 345
Matt Wette

Test Systems Engineering Group, Section 341
Dennis McCreary

Edited by Brad Hines

This report was prepared by participants from the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Revision Log

Version	Revision Note	Date Published	Date Reviewed
1.1	Incorporated comments from Document Review	11/17/1999	11/16/1999
1.0	Released	10/27/1999	10/26/1999

Introduction

The authors of this report have recently returned from the 10th annual Embedded Systems Conference in San Jose, California. A broad array of topics were covered in the tutorials and classes at the conference, and vendors displayed a wide variety of products. The conference was attended by over 16,000 people, and is the single biggest event in the embedded industry each year.

We learned a lot of noteworthy things at the show, and felt it worthwhile to make this report of the highlights of what we saw and learned. The first section of this report is a high-level summary of what we all saw, with more details on each of the topics in the later sections.

Each attendee received the proceedings of the conference on CD-ROM, which means that we have available papers covering most of the topics summarized in this report for anyone that would like to read them. Contact any of the authors of this report if you'd like more info on a particular topic. Also, Dennis McCreary has arranged for the CD to be available online on huey for the next few months; just type "esc99" at the Unix prompt to view the CD contents. Alternatively, you can borrow a CD from one of the authors.

Note to Managers *(M)*

This report contains a lot of technical content, but it also contains substantial amounts of information of interest to managers. If you're a manager and just want to browse the parts of the report having to do with management, look for the *(M)* indicator in the right margin of the section heading, such as is in the heading for this section. Items marked with a small m *(m)* may be of interest to those directly managing software projects.

That is, if you're a manager of a project which has software or embedded systems as a component, I think you'll want to be aware of at least the *(M)* items. If you directly manage a software or embedded development task, I think you'll also want to be aware of the *(m)* items. The last of the *(M)/(m)* items appears on page 8 of this report.

Summary

Java *(M)*

Believe it or not, Java for embedded applications is making a huge splash in the industry. A number of vendors are now supplying small-footprint Java Virtual Machines for embedded processors, and there are even a number of hardware Java machines in silicon that you can buy.

JPL in the news *(M)*

Several attendees were regaled on several occasions with examples of software failures brought to you by JPL, with Mars Pathfinder playing the starring role (2 separate failures that were popular), and Mars Climate Orbiter mentioned as a possible software failure. The Ariane 5 failure was also mentioned, but less popular. Software people like to illustrate their points with really spectacular failures.

Code Reviews *(M)*

Every class I am aware of that covered management of embedded projects ended up quickly focusing on management of software. Each of these classes homed in on the importance of code reviews in delivering software on time and controlling maintenance costs.

Documentation and Bugs *(M)*

Classes stressed that bugs should be fixed when found, that lists of known bugs should not be allowed to continue to grow, and that all system failures should be recorded. Documentation should be written, at least the first cut, before significant coding takes place.

Performance
A number of attendees learned about coding for performance, including optimization of virtual functions, avoiding temporary and dynamic objects, references, overloading new, and manipulating registers from C/C++.

C++ *(m)*
Much embedded development is still done in C, but there is continuous migration into C++ and greater levels of object orientation. The Embedded C++ standard (EC++) is a sort of compromise that provides many useful features of C++ while losing some of the bulk of the full language.

Hardware Design
People learned about a number of logic design issues and tools.

Tools
GDB is now supporting "tracepoints", which are a way of allowing you to step through realtime code without actually stopping it. Execution data is recorded while the code executes, allowing you to "replay" it later.

Hardware
500 MHz PowerPC 750 boards are now available. "System-on-a-chip" is becoming a big deal for folks designing small, smart, all-in-one devices. You can buy some really small stuff, 486 computers and hard drives that fit in your shirt pocket, etc. There is hardware to translate between just about any bus you want now, it seems, VME, PCI, CPCI, 1394,... Debugging with hardware emulators is becoming tougher as chips get faster, so debugging facilities are moving on-chip.

Open Source *(m)*
The open source model, where the source code is free, with the option to pay for support, is increasing in popularity. In addition to the Gnu packages, there are now open-source real-time operating systems (uC/OS-II, an 8-bit OS, VRTX from Mentor Graphics, and ECOS from Cygnus Support), and Linux is coming on strong as an embedded and development platform and garnering lots of support from hardware vendors,

Realtime OS Technology
The Priority Ceiling Protocol is an exciting twist, one that is getting more attention recently, that completely eliminates the possibility of priority inversion problems (such as were experienced by Pathfinder) and the dreaded deadlock (!) and allows predictable design of realtime systems. Vendors are starting to support OSEK, a new microkernel standard for small RTOS's. There was also a talk on scheduling technology from the 1970's to today.

Microsoft
Windows CE is really targeted towards handheld devices with displays and that sort of thing, but it does have relatively impressive realtime performance and is a viable platform for many applications, including the sorts of things we do.

Busses and Protocols *(m)*
CAN (Controller Area Network) and TTP (Time-Triggered Protocol) are being used in medium-speed (1 Mbit/sec) applications such as automobiles. TTP seems to show special promise as far as being able to build deterministic and predictable realtime systems with fewer surprises during integration. I2O is a Wind-River-backed I/O idea for putting smart processors in I/O devices, which has benefits for hardware vendors who must supply drivers to their customers on multiple OS's. Several folks attended a 1394 (Firewire) class and now have a pretty deep understanding of it. USB is sort of lower-speed incarnation of Firewire. Both USB and Firewire are about to undergo a spec update, which has the potential to cause backward compatibility problems. It's important to be aware of the software environment used with the bus in a particular system as well, since many standard software layers (e.g. SBP-2 over Firewire) add nontrivial overhead.

Standards *(m)*

XML seems to be on a roll, and is positioned to become the common data interchange format of the future. Many tools, including your desktop office suite, are adopting XML or at least supporting it.

Miscellaneous – "Linux is the best documentation I know of for device driver writers," asserted one speaker. So if there is already support for a board you have in Linux, run, don't walk, to the Linux sources and steal away.

Of Note - The line for the men's room was consistently longer than the line for the women's room.

Detailed reports

Java – *Hines, Irwin* *(m)*

Industry interest in Java seems to be coming from several directions. There are a lot of realtime Java products available, from compilers, to Java-to-C++ converters (so that you can then compile), to virtual machines for embedded use, to Java processors that execute Java byte-codes directly.

One source of interest is the commonly held opinion that Java is an easier language to learn than C++. Also, since Java does not have pointers and eliminates a whole class of programming errors related to memory management (seasoned programmers learn how to avoid these pitfalls, but novices are often victims of them), people are enthused about the ability to eliminate these errors. Memory leaks can be particularly problematic in embedded systems, which sometimes are required to run for weeks or months at a time.

Another bonus is portability, obviously. You test your code on a workstation and it can run on your embedded system without recompiling. I can also imagine things like our Apparent Place calculations, which are currently written in C++, being written in Java. Then, when a bug is found and corrected, our various field installations can just grab the new code and insert it into their system. No need to recompile, rebuild, etc. The new stuff just starts working.

As far as performance, there has been significant technical progress that makes performance much less of an issue. The common gut reaction is that a language that does garbage collection is unfit for realtime. However, there are now garbage collectors available that are compatible with realtime use, using techniques such as incremental garbage collection. One advantage to garbage collection is that heap fragmentation, which can be a particular concern again for systems that have to run for very long uptimes, is not an issue.

A second performance concern has to do with the fact that Java is interpreted instead of compiled. There are a couple of solutions to this available now. There are now compilers that compile Java directly into native machine code for performance-sensitive applications. These applications then tend to be only about 50% slower than the equivalent C++ program. The performance difference is largely due to Java's runtime error-checking (e.g. array bounds, etc.), and most compilers offer the option to turn this error-checking off for performance-sensitive applications.

For people that are developing embedded code for small devices, where designers typically develop their own hardware as well as software, hardware Java machines are an excellent solution to the performance problem.

C++ in Embedded Systems – *Wette*

Matt attended a talk and another "shop talk" on using C++ for embedded systems. The speaker was Don Saks, who has a company providing support for C++. In his talk the speaker overviewed some features of

C++ and discussed the trend in embedded systems and some ways to migrate to C++. Two features that are of current concern in embedded systems are (1) exception handling and (2) runtime type information.

There is also a Embedded C++ standard (EC++) that is supported by many vendors. The official C++ standards committee now has a working group on performance issues. However, no recommendation was given for EC++ as (1) many features which don't have performance issues were removed (e.g., templates, namespaces) and (2) the standard is incomplete (e.g., there is no specification to cover failure of operator new). Another bit of useful advice was that the EC++ Standard Library can provide a useful replacement for the more bulky Standard C++ Library. In summary, it seems that people are not making the choice of whether to convert to C++ but when (though Java is making a splash). For more info on EC++, see <http://www.caravan.net/ec2plus/>.

Also, after the conference Matt talked to Kenny Meyers of MDS about C++/Java issues. He showed Matt a cool book by Stan Lippman called "Inside C++ the Object Model" which talks about internals of C++ code layout. Matt went out and bought it at Barnes and Noble -- looks good. Also, Kenny said that Stan is coming out with a book on performance.

Realtime Operating Systems and OS Technology – *Wette, Hines, McCreary*

One talk, "A Survey of Task Schedulers" provided an overview of schedulers from the 1970's to today. The talk "A Modern Standard for Super-Small Kernels ..." provided details on the new OSEK standard for small RTOSs, which many vendors (including WRS) are starting to support. A talk entitled "Inside Real-Time Kernels" covered the author's popular, open-source 8-bit RTOS called uC/OS-II.

The best talk of this lot was "Predictable Real-Time System Design" given by a speaker from TimeSys. He gave an overview of realtime systems issues (priority inversion, RMA etc) and touched a little on CORBA and its implications. Matt has contacted these folks to come give a talk, possibly the morning of Nov 3. The current feeling seems to be that the Priority Inversion Protocol for schedulers is not as reliable as Priority Ceiling Protocol.

Priority Ceiling Protocol

Most OS's don't quite support this fully yet, but it will likely be coming soon, since it solves some nasty problems (priority inversion and deadlock) so beautifully. The idea is this.

Priority inversion occurs when a low-priority task owns a resource that's needed by a high-priority task. The high-priority task has to wait on the low-priority task to finish using the resource before it can proceed. OS's (VxWorks, for example) provide a way to get around this with priority inheritance. In this scheme, as soon as the OS detects that the high-priority task is waiting on a resource owned by the low-priority task, it temporarily boosts the priority of the low-priority task (the task "inherits" the priority of the higher-priority task temporarily). Now the low-priority task will run and release the resource and the high-priority task can then proceed.

However, this doesn't solve all problems of this type. Deadlock, where two tasks each require two resources, but are each blocked waiting on one of them, which is already owned by the other task, is a serious problem in some systems.

The Priority Ceiling Protocol slightly changes the idea of priority inheritance to eliminate both priority inversion and deadlock. Basically, the OS (or you, manually) keeps track of all the potential users of a resource (e.g. of a semaphore), and marks that resource with the priority of the highest-priority potential user. That is, the resource is assigned the "priority ceiling" of all the possible users' priorities.

Then, when a task acquires the resource, its priority is instantly boosted to the priority ceiling for the resource and remains there until it releases the resource. Since there is no other contender for the resource with higher priority, there will never be any contention for the resource. A moment's thought reveals that this solves the problem of deadlock, too.

It's a very cute and clever solution with few drawbacks. The main drawback is the repeated boosting of low-priority tasks to high priority even when there is no contention for the resource.

However, an advantage of PCP is that it makes the system' realtime performance analytically predictable. There are more details on this in the paper on PCP in the conference proceedings.

Time-Triggered Protocol – *Wette*

TTP/C (Time-Triggered Protocol) is a standard for bus-based communication distributed by TTTech in Germany. It seems to be big in the auto industry and is a complement / alternative to CAN (Controller Area Network).

The main difference between CAN and TTP is that TTP is "state based" rather than "event based". Nodes (repeatedly) transmit their current state on a periodic basis rather than sending single messages. For example, if a switch is turned on in the network the associated bus message would not be "switch turned on" but "switch is off [wait] switch is on [wait] switch is on ...".

All bus traffic is completely deterministic, allocated a priori and configured in each node as a dispatch table. The advantage here is that TTP is robust to lost messages. There is more. Matt has a copy of the spec if anyone wants to look. Also, more on TTP is available from <http://www.tttech.com>. One of the coolest things that came up in the talk is that BMW has a prototype SUV with steer-by-wire, and it works!

Code reviews - *Hines* *(M)*

There appears to be no debate remaining in the industry. Code reviews appear to be uncontested as the way to improve productivity and reduce cost. The claim is that studies show that code review is *20 times* more efficient at finding bugs than debugging is. A full day of code review with 4 people can be a bit tedious and seem unproductive, but many times a particularly insidious bug can take a skilled programmer two weeks to track down. If the full-day code review finds only three or four bugs, it quickly earns its keep. Typically, the reviews are significantly more effective than that.

Reviews are also credited with being especially helpful during the maintenance phase, after the original programmers have moved on. Code prepared for review is often in better condition, with better documentation, than code that has simply been developed and tested by a single developer until it works.

It seems clear that, within a couple of years, any organization that does not implement code reviews as part of its standard development practice is going to look like a dinosaur relative to the rest of the industry.

Iterative development - *Hines* *(M)*

Iterative development was also a universal theme. Nobody believes in waterfall anymore. The idea is to get a skeleton of the whole application running as quickly as possible, and then begin slowly filling it in with functionality. The idea of testing daily (or at least weekly) was also strongly endorsed.

Squash bugs <u>now</u> - *Hines* *(M)*

The purpose of code reviews and daily regression testing is to catch bugs as soon as possible. There was a universal thread through all the management talks that bugs should be fixed as soon as they are detected. Pushing to make a delivery date and making the delivery with a list of known bugs outstanding as liens was universally declared to be a poor practice. Stop, fix the bugs now. The bugs should be fixed while the code is fresh in the programmers' minds, and while the bugs are repeatable. Latent bugs can interact with other bugs to create extremely difficult-to-diagnose problems.

A related point is that all failures of a system should be recorded. Often, when developers encounter anomalous behavior, they just "try it again." Recurring bugs can slip through the cracks this way, as developers repeatedly ignore a serious problem because it's simply an annoyance to their problem at hand. It's important to record every failure of the system in order to make sure this kind of thing doesn't slip through the cracks.

Pay attention to the error cases – *Hines* *(M)*

The loss of the first Ariane 5 is the classic example of an error case that was trapped by the program and handled exactly as the programmers intended, yet led to system failure anyway. In the Ariane 5, the Inertial Measurement Unit experienced a numeric overflow of a 16-bit value due to the larger forces, etc. in the Ariane 5 than had existed in the previous-generation rocket, in which the IMU had been successfully used.

The software trapped the error, and dutifully shut down the IMU and switched to the backup unit. Of course, the backup unit immediately experienced the same problem and was shut down by software as well. After that, the rocket didn't fly so well.

Example after example was presented of systems that failed because of incorrect behavior in the off-nominal case. These cases are often not specified well, or left up to the programmers to handle, and the specific behavior in these cases is not documented as part of the documentation of a software module. As the examples show, this can have catastrophic results, especially when it comes time to reuse the software. Complete specification and documentation of a module's behavior is important in order to avoid these kinds of problems.

I20 – *Hines, Wette*

Intelligent I/O (I2O) is a protocol/design scheme that calls for I/O devices to have an embedded processor that responds to a particular message-passing protocol. The idea is that this achieves some level of device independence for software drivers. I2O specifies standard interfaces to about 20 different types of devices, including printers, modems, etc.

The idea, then, is that the OS vendor just supports "I2O fax/modem", rather than having to support devices from hundreds of different vendors. The vendor then places his value-added software on the embedded processor in the I2O device. When a driver update is needed, the vendor only has to update one driver, instead of one for every OS platform that the device works with.

There are currently some legal harangues surrounding I2O, having to do with the fact that it's a Wind River initiative and they've got their fingers into it in some slightly inextricable ways.

It's not clear how applicable I2O is to us directly at the moment, but it is reasonably possible that as time goes on, more and more of the devices/boards we buy will include I2O support, and VxWorks will provide I2O drivers, and this will be a convenient way to access certain common hardware types, such as D/A converters, making it much easier for us to develop drivers.

XML – *McCreary*

XML is the wave of the future. It is ideally suited for exchanging information between vastly different applications with different needs (HTML is not). XML is expected to be the common data interchange format of the future. XML has a simple universal syntax, flexible data model, multiple namespaces and is being adopted by many tools across many industries (Excel, Word, Office, Internet Explorer, and Oracle among others). Transformation programs exist which will translate XML to XML, HTML, SVGML and MathML.

Debugging Technology - *McCreary, McKenney, Wette*

Several companies are coming out with debuggers which allow visibility into realtime code without changing execution timing, at least to some extent. These tools, including a new version of GDB, work by making a record of what actually happened during the realtime execution and then allowing you to step through the code, examine variables, and so on, using saved information from the trace buffer.

Xray Debugger from Mentor Graphics allows "synchronous breakpoints," which allow the execution of several tasks to be simultaneously suspended when a breakpoint is reached in a single task.

Coding and Debugging Techniques - *Regehr*

There are a few techniques that can usually be used to improve the performance of C++ code. Passing objects by reference-to-const avoids a copy constructor call, and the creation of temporary objects can sometimes be avoided by using multiple unary operators instead of binary operators. Performance gains can sometimes be achieved in programs which do a lot of memory allocation/deallocation by overloading 'new' with a 'placement new' function which allocates memory from a pre-allocated pool in fixed-size blocks in order to mitigate the variable overhead and memory fragmentation which can be caused by the standard new' function.

For debugging interrupt service routines and other hard realtime events, you can use techniques similar to what we have used in the past, such as reserving some parallel output bits for debugging, and using an oscilloscope and an external counter to monitor the frequency of interrupts, the fraction of CPU time spent in ISR's, and to detect missed interrupts. Also, monitoring the size of the stack using a stack monitor is considered a good idea.

Flash Memory - *McCreary, Regehr*

Flash memory is available in gigabyte sizes, and one of the companies that makes it claims it is rad hard and thus might be suitable as a substitute for a disk drive on a spacecraft. Drivers exist for many operating systems to make a bank of flash memory look like a disk drive. We have to do a little more research to determine how these capabilities relate to VxWorks flash file system that flew on Mars Pathfinder.

Meta-Information

This section doesn't contain as many details on things we learned, but it is a report from various folks on a list of topics that they learned about. We include it here so folks can see what else we learned that we didn't necessarily write up in this report. If you are interested in any of the things mentioned here, contact the person who made the report for more information, and he or she can point you to the appropriate papers from the conference.

Beth McKenney

I learned some creative ways to reduce runtime overhead in C++, such as (a) forcing virtual function calls to be non-virtual in time-critical parts of the code, (b) convincing the compiler not to create superfluous temporary objects using a variety of techniques, and (c) avoiding dynamic creation/deletion of objects where possible.

Also learned a few tricks for representing and manipulating hardware registers using C/C++ (with the caveat that some of the techniques may be compiler- or system-dependent).

I learned about the differences between EPLDs and FPGAs, as well as some of the features to look for when selecting a chip for a given application. Heard some good treatment of various FPGA design issues, such as hold time violations, bus contention, glitches, one-hot encoding, and the rare cases where asynchronous logic might possibly be advisable.

Also learned a bit about FPGA design tools, including key differences between the Altera and Xilinx compiler strategies.

I learned that it is possible to insert a "stub" of the GDB debugger into an embedded system, and set it up to record trace variables at desired locations ("tracepoints") during runtime, minimizing the impact on the realtime system behavior. The variables can then be read back at a later time using the GDB interface with a few simple extensions. [This has been done by some people at Cygnus, but we could probably do it ourselves in the time it will take Cygnus to figure out how it wants to market their stuff.]

[Matt] If this stuff is part of the GDB sources, then the developer versions of that stuff is available. In fact, Cygnus provides access to most of the CVS development trees (i.e., you can do things like "cvs checkout" the gdb development tree).

Matt Wette

I visited several vendor displays and got some insight on the state of the art in compilers, debuggers, and other such tools. I also visited the SBS display which provided some up-to-date information on I/O capability we'll be needing for the SIM testbeds. Processing for the simulation in the SIM Flight System Testbed should not be an issues. Vendors (e.g., Motorola) are now coming out with 500MHz 750 processors for VME and CPCI.

There was a talk on MS Windows CE focused on changes to the upcoming version 3.0 that developers needed to be aware of.

Phil Irwin

Here's some stuff that caught my attention:

Firewire/USB:

- These two busses are very similar, USB is a lower speed version with a higher max node count
- Both are about to undergo a spec update, which could cause incompatibilities in devices. Firewire especially so because there are some new rules that could make it very difficult to interpret the spec.
- USB is better for input devices...Firewire for storage, video...

C++:

- Dan Saks gave a great talk on reducing runtime overhead in C++
- Many of the things he mentioned, we already do. He even went as far as to fake out the virtual table and call a function directly if it were to be called in a long repetitive loop.
- He also talked about temporary objects, overloading new, placement new, and using references to your advantage.

System-on-a-chip:

- No speaker really gave me a good handle on this but it sounds like it's much better for super-high quantity production (definitely not our bag), but it's a really cool concept.

Hardware

- You can buy a 486 computer on a 2"x3" board, complete with ISA bus
- You can also get a multi-hundred MB hard drive that fits on a L2 PC card.
- SBS has some neat stuff for bus conversion (VME-PCI, VME-IEEE1394, etc...).

Software:

- There are a few hardware companies getting excited about Linux; software companies don't like it, especially ones that sell operating systems.
- There are bazillions of software development tool companies. I didn't get too excited about any of them. If we want to increase our debugging capabilities, maybe we should think about hardware debugging.

Graham Hardy

Since RTOS's are relatively new to me I gravitated to the talks about the nature of Real-Time Kernels and Multitasking Design. The things I learned were:

1. The nature of the following: Linear Code (foreground/background) vs. Multitasking, Multitasking vs. Multithreading, Prioritization, Determinism, Preemptive vs. non-Preemptive Scheduling
2. The structure of a RTOS (e.g.: Task States, TCB's, Ready Lists, System Ticks, Interrupt-Kernel interaction, Wait Lists, Kernel Services, Stack Checking, etc)
3. The considerations and tradeoffs of choosing Build vs. Buy with respect to RTOS's.
4. Getting a Digital ASIC designed and built is pretty straightforward though expensive (also the tradeoffs of Gate Arrays vs. Standard Cells).
5. Modern State Machine theory is bloody complicated!!!

Dennis McCreary
I took the following classes. Here are my comments on each.
100 TCP/IP Networking

202 The Integration of Embedded Web Technology with XML
Already discussed in "Detailed Reports" above.

242 Debugging ISRs
"If debugging is the process of removing bugs, then programming must be the process of putting them in."
"Trust nothing, assume nothing, presume failure."
This talk presented a philosophy on the design, implementation and testing of ISRs. This is a good paper worth reading if you are interested in ISRs.

260 Flash Memory Technology and Techniques
Flash memory retains information that is written to it even after the power is turned off. It will retain the information for 10 years plus. While it can be programmed at the byte level, an erase operation erases an entire block. This behavior must be taken into account when designing applications that use it. With repeated use, the memory can degrade. It is still usable, but takes longer to program or erase. Several software vendors have written packages that emulate disk drives in flash memory. If a sector previously written to the device needs to be modified, the old sector is marked dirty and the new changed sector is written to a different location. When all the sectors in a block have been marked dirty, it can be erased and reclaimed. Devices about the size of a half height disk drive exist which hold several gigabytes of flash memory.

301 Architecture of Device I/O Drivers, Parts 1 & 2
Some specific strategies for particular types of device drivers.

341 Device Drivers
Good overview of issues involved in writing device drivers. "Linux is the best documentation I know of for driver writers."

368 Memory Management
Discusses memory management issues on the stack and the heap. Shows how malloc works and discusses alternate schemes for pools or partitioning memory to reduce fragmentation problems. Talks about third party software memory management issues and automatic garbage collection. Presents a smart pointer class for C++ which handles memory allocation.

403 Reducing Runtime Overhead in C++
Excellent talk. Discusses optimization issues with virtual functions, parameter passing, ++i vs. i++, replacing the new and delete operators and using placement new among others.

423 Representing and Manipulating Hardware in Standard C and C++
Excellent talk. Standard C and C++ are excellent choices for devices that use memory mapped I/O. With the use of minor deviations, they are also a good choice for devices which do port I/O. For architectures that provide special I/O instructions dedicated to particular devices, they are a poor choice. Discusses the use of "const", "volatile", and "sig_atomic_t". Talks about various ways to describe the memory mapped

hardware including pointers to hardware registers, casting, masking, registers as objects, tags vs types, register pairs, and write only registers. Also discusses problems with optimization.

444 Fundamentals of Firewire, Parts 1 & 2
Good overview of what Firewire is, where it is headed and how it works.

509 The Heisenberg Debugging Technology
The people at Cygnus have added trace points to the GDB debugger. The advantage of trace points is that data is collected and buffered internally while the program is running and is processed offline after the run is over. This allows the program to run much closer to real time while it is collecting the debugging information. The modifications to the GDB debugger required to do this are publicly available. On the target side, a GDB stub is required. This stub has been written by Cygnus, but they haven't decided what they are going to do with it yet. Using this debugging technique, it is possible to have unattended collection of debug data over extended periods of time on the order of days or weeks, depending, of course, on the size of the internal buffer allocated and the amount of debug data being collected. When processing the collected data, you can replay it multiple times and do calculations based on any of the values collected. This would be very useful if we could get hold of the stub or write our own.

567 How Scripting Adds Value to Embedded Systems
With the rise in processing speed, memory size, and complexity of embedded systems, onboard scripting is going to become increasingly more important.

John Ousterhout, who gave this talk, believes that TCL is the best choice for a scripting language. (I second that opinion.) TCL is easily extensible, can be embedded and used to produce an application command language, is easy to hook to anything, has great GUIs, is cross platform, and is open source. It is ideally suited to integration applications where several separate applications must be tied together. An informal study indicated that it was 3 to 10 times faster to develop an integration application using TCL vs. using Java, C or C++. In particular, John feels that Java is not a good match for embedded programming and presents a list of reasons why he feels this way.

I also picked up hardcopies of the presenter's slides for the following classes that I did not attend:
243 How to Size Message Queues
401 Inside Real-Time Kernels, Parts 1 & 2
529 Debugging Tools, Trends and Tradeoffs in an Embedded Design Project
566 Moving from a Standard Programmable Interrupt Controller to an Open Programmable Interrupt Controller

I also spent some time talking to vendors of hardware and software debuggers and flash memory.
Flash memory is available in gigabyte sizes and one of the companies that makes it claims it is rad hard and thus might be suitable as a substitute for a disk drive on a spacecraft. I asked them to supply me with the names of people who use their product for disk drives on cards which can be installed in a VME cage.

I saw demonstrations of several hardware and software debuggers.

Mentor Graphics makes Xray Debugger, a software product which allows synchronous breakpoints. You can request that when a particular breakpoint occurs in a particular task, that several other tasks running on the same or different targets be breakpointed at the same time. This sounds like something that would be useful to us.

Green Hills makes a software debugger called Metro 2000 which it turns out that the FST has a license for. I plan to take a look at it.

Embedded Support Tools Corp makes several hardware debuggers which were impressive. They support several hardware interfaces including JTAG, and in one setup, you can do things like breakpoint the hardware when a piece of software outside a particular range of addresses attempts to store into an address

inside a particular range of addresses. This would allow you to trap a program that was walking all over memory while not stopping for legitimate uses of that memory. Another advantage of the hardware debugger is that you can examine memory even if the machine has locked up. With software debuggers, if this happens, you have to start over and try again.

Tektronix demoed a nice logic analyzer with lots of memory so it could buffer up lots of data. We have an earlier model here in the STB3 lab and I plan to find out more about how it works and what it can do.

Cygnus has upgraded the GNU debugger to include tracepoints which allow debugging closer to realtime. The code for this in the debugger is publicly available but they haven't yet decided what to do with the stub which is required on the target side. For a fee, we might be able to get them to install it here.

Martin Regehr

Cristopher Leidigh of American Power Conversion gave a fairly useful tutorial on TCP/IP which included details of ARP, IP, and TCP. He performed live demonstrations of various concepts using communication between his laptop computer and other networked devices, and a software package called Observer which displayed transmissions at the byte level and interpreted them at the level of different layers in the TCP/IP stack. My Tran of Motorola discussed the OpenPIC (Programmable interrupt controller) as an alternative to the older 8259A interrupt controller. The OpenPIC supports multiple processors and inter-processor interrupts, and allows interrupt priorities to be set using registers on the chip. Apparently HDL specifying the chip is available for free and manufacturers who sell this chip or incorporate it into other chips need not pay license fees.

As discussed earlier, Dan Saks of Saks and Associates gave several talks on C++ and C, including a good review talk on templates and a talk on improving the speed of C++ code, and Jack Ganssle of The Ganssle Group discussed various techniques for debugging ISR's

Bill Grundmann of Intel discussed their flash memory technology. Apparently the main advantage of flash memory is that it is non-volatile; disadvantages are slow speed and the fact that erasing can only be done a large block at a time. He discussed page mode access for speeding up reads, and allocation algorithms for mitigating the inconvenience of erasing. Windows CE drivers are available which will make a bank of flash memory look like disk space to the operating system.